August 1988

UILU-ENG-88-2245
ACT-96

COORDINATED SCIENCE LABORATORY
College of Engineering
Applied Computation Theory

HIERARCHIES AND SPACE MEASURES FOR POINTER MACHINES

David R. Luginbuhl
Michael C. Loui

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-88-2245 (ACT-96)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) 1101 W. Springfield Avenue Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING / SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

Hieracrchies and Space Measures for Pointer Machines

12. PERSONAL AUTHOR(S)
Luginbuhl, David R. and Loui, Michael C.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	August, 1988	30

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Invariance, pointer machine, random access machine, space complexity, time complexity, Turing machine

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

We establish several time and space results for Schönhage's pointer machine (PM), including space compression, time hierarchy, and space hierarchy. We also present two definitions of space complexity for PMs, and we consider how each space measure affects the equivalence of PMs to the more classical models of computation.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

Hierarchies and Space Measures for Pointer Machines

David R. Luginbuhl *
Department of Computer Science and
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, IL 61801

Michael C. Loui
Department of Electrical and Computer Engineering and
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, IL 61801

July 29, 1988

*Supported by: Air Force Institute of Technology, AFIT/CIRD, Wright-Patterson AFB, OH 45433

Abstract

We establish several time and space results for Schönhage's pointer machine (PM), including space compression, time hierarchy, and space hierarchy. We also present two definitions of space complexity for PMs, and we consider how each space measure affects the equivalence of PMs to the more classical models of computation.

KEYWORDS: Invariance, pointer machine, random access machine, space complexity, time complexity, Turing machine.

List of Symbols

symbol	*name*
α	lowercase alpha
β	lowercase beta
Δ	uppercase delta
δ	lowercase delta
γ	lowercase gamma
$\mathcal{L}$	calligraphic style uppercase ell
μ	lowercase mu
ν	lowercase nu
ϕ	lowercase phi
ψ	lowercase psi
ρ	lowercase rho
Σ	uppercase sigma
σ	lowercase sigma
Ω	uppercase omega
$\in$	element of
$\subseteq$	subset of
$\subset$	proper subset of

1 Introduction

In 1980, Schönhage (1980) described in detail the Storage Modification Machine, also called the pointer machine (PM). He showed that in terms of time complexity this computational model is as powerful as a Random Access Machine (RAM) (Cook and Reckhow, 1973) and a multi-dimensional Turing machine (TM), and he asserted that the PM is more useful for measuring lower-order time complexity.

Some results have been obtained in recent years using the PM (Halpern *et al.*, 1986; Tarjan, 1979). For the most part, however, this model has received little attention until recently (Gurevich, 1988; van Emde Boas, 1987). Moreover, most of the work on PMs has focused on time complexity. Only Halpern *et al.* (1986) and van Emde Boas (1987) address the issue of space complexity.

In this paper, we present several results concerning both time and space complexity of PMs. These results indicate that, in many ways, PMs are similar to TMs and RAMs.

We begin with a review of the definition of a PM, and we present two possible definitions for space complexity. We show that space compression is possible with this model under one of these space measures. We also give time and space hierarchy theorems for PMs. With respect to the time and space hierarchies, PMs are similar to TMs and RAMs. Our final results demonstrate how the definition of space complexity of PMs affects the equivalence of PMs to other models of computation.

After we obtained these results, we discovered that van Emde Boas (1987)

had independently investigated the issue of space complexity for PMs. In fact, he had already obtained the space bound result of Theorem 5.1. Our work, however, represents an improvement: our simulation yields a significant speedup in time, and we show that the result is optimal in terms of space complexity.

2 Pointer Machines

We propose the following as a "standard form" for PMs. It is a hybrid of the PMs described in (Halpern *et al.*, 1986) and (Schönhage, 1980).

As in (Halpern *et al.*, 1986) and (Schönhage, 1980), the Δ-*structure*, which provides the storage for the PM, is a directed graph consisting of *nodes* (vertices) and *pointers* (edges). Each node has a finite number of outgoing pointers, and each pointer from a node has a distinct label. The labels are symbols from the pointer alphabet Δ. At any time, one node, designated the *center*, is used to access the Δ-structure. We refer to the center node as x_0.

The instantaneous configuration of the Δ-structure is described by the pointer mapping $p_\delta : X \rightarrow X$, where X is the set of nodes, and $p_\delta(x) = y$ means the δ pointer from node x points to node y.

The PM also has a separate read-only input tape containing symbols from an input alphabet Σ. For simplicity, we consider only PM acceptors. With the addition of a write-only output tape, we could also consider transducers.

A PM has a finite sequence of program instructions, each with some distinct label. The following are allowable instructions:

accept. Self-explanatory. Computation halts.

reject. Self-explanatory. Again, computation halts.

create δ, where $\delta \in \Delta$. Create a new node x' in the Δ-structure. $p_\delta(x_0)$ is set to x'. For all $\gamma \in \Delta$, $p_\gamma(x')$ is set to x_0.

center δ, where $\delta \in \Delta$. Make the node $p_\delta(x_0)$ the new center.

assign $\delta := \gamma\nu$, where $\delta, \gamma, \nu \in \Delta$. Change the δ pointer from the center. It should now point to the node reached by following the pointers γ then ν, starting from the center; that is, $p_\delta(x_0) = p_\nu(p_\gamma(x_0))$. We also allow ν to be a null pointer symbol, in which case we would have $p_\delta(x_0) = p_\gamma(x_0)$.

if $\gamma = \delta$ **go to** μ. If $p_\gamma(x_0) = p_\delta(x_0)$, then pass control to the instruction labeled μ. Otherwise, execute the next instruction.

if *input* $= \sigma$ **go to** μ. If the input symbol is σ, then pass control to the instruction labeled μ. Otherwise, execute the next instruction.

move ρ, where $\rho \in \{$**left**, **right**$\}$. Move the input tape head one square in the direction indicated by ρ.

The PM starts with the input head on the leftmost nonblank input symbol and one node in the Δ-structure. We call this node, which is the center when computation begins, the *origin.*

The *time* consumed by the PM is the number of instructions executed before halting. We consider two space measures. Define *mass* to be the

number of **create** instructions executed before halting, i.e., the number of nodes created during execution. Mass was introduced as a measure of space in (Halpern *et al.*, 1986). The *capacity* of a computation is $dn \lg n$ ($\lg = \log_2$), where n is the number of nodes created, and d is the number of pointers per node ($d = |\Delta|$).

The idea for considering capacity as a space measure comes from Borodin *et al.* (1981). With n nodes there are at most n^{dn} possible configurations of the Δ-structure. In (Borodin *et al.*, 1981), control space (capacity) is defined as $\lg(Q)$, where Q is the number of possible configurations, so in the case of PMs, defining capacity as $dn \lg n$ is reasonable.

We define a function $T(n)$ (respectively, $S_m(n), S_c(n)$) to be *time- (respectively, mass-, capacity-) constructible* by a PM if there is a PM such that, for every input of length n, the PM halts in time $T(n)$ (respectively, mass $S_m(n)$, capacity $S_c(n)$).

We say a PM has *time complexity* $T(n)$ if for every input of length n, the maximum execution time for the PM is $T(n)$. We define *mass complexity* and *capacity complexity* similarly. Complexity classes corresponding to time, mass, and capacity for PMs are, respectively, $\text{TIME}_{PM}(T)$, $\text{MASS}(S)$, and $\text{CAPACITY}(S)$.

3 Space Compression

Our first result is a space compression theorem for PMs.

Theorem 3.1 *For every constant $c > 0$, every pointer machine with mass complexity $S_m(n)$ can be simulated by some pointer machine with mass complexity $cS_m(n)$.*

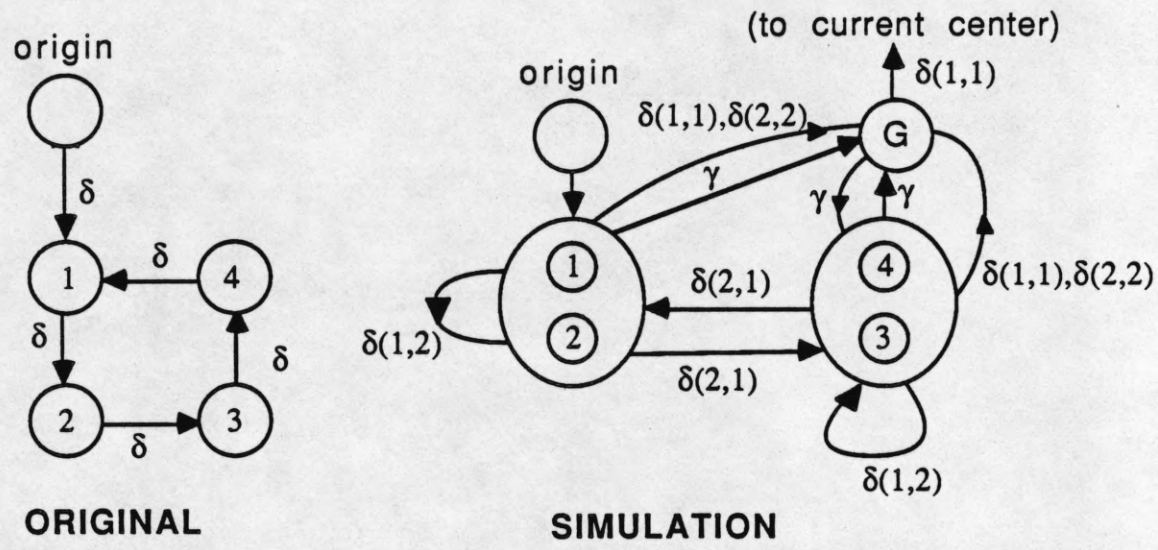
Proof. We show the case where $c = 1/2$. Consider PM A having mass complexity $S_m(n)$. We design a PM B that simulates A and has mass complexity $S_m(n)/2$.

For B to compute with half the number of nodes of A, we encode two nodes of A into one node of B with the addition of several pointers. For every δ in the pointer alphabet of A, the pointer alphabet of B has $\delta(1,1)$, $\delta(1,2)$, $\delta(2,1)$, $\delta(2,2)$. Each node in B corresponds to a pair of nodes in A. The ordered pairs in the pointer notation indicate the original source and destination nodes.

We also create one node (called G) to hold "useless" pointers. And we need a pointer γ that points to G from any node, so G is always accessible. The γ pointer from G always points to the last node created.

We establish the correspondence between nodes of A and nodes of B as follows. Call a node in B a *node-pair* to distinguish it from the pair of nodes in A to which it corresponds. Designate the older node in a pair of nodes in A as node 1 and the other as node 2. If the δ pointer of node 1 in a pair corresponding to node-pair a in B points to node 1 in a pair corresponding to node-pair b, then $p_{\delta(1,1)}(a) = b$ and $p_{\delta(1,2)}(a) = G$. Other cases are handled similarly (see Figure 1).

Since we are working with node-pairs in B , we need to designate where the center is within a pair. The structure of B will tell us whether a node-



(Nodes are numbered in the sequence they were created)

Figure 1: Reducing the number of nodes by 1/2

pair contains the center of the structure of A by identifying that node-pair as the center. Then we can use a pointer ϕ and two additional nodes H_1 and H_2 in B to tell us whether the center is node 1 or node 2 by having ϕ point to H_1 or H_2, as appropriate.

Since we may occasionally need to make G the temporary center to check its pointers, we can use $\delta(1,1)$ to point from G to the actual center node-pair. In this way, we can always re-establish the center in B.

We initialize B by creating nodes G, H_1, and H_2. After we set their pointers appropriately, we are ready to simulate A.

Rather than describe the simulation of A in tedious detail, we discuss how to simulate one instruction, **create** δ. The other instructions are simulated analogously.

To simulate **create** δ, we first find the last node-pair created by following the γ pointer of G. Call this node-pair a. We then determine whether node-pair a corresponds to a single node in A (if an odd number of nodes have been created in the execution of A at this point) or to an actual pair. If $p_{\delta(2,1)}(a) = G$ and $p_{\delta(2,2)}(a) = G$, then the node to be created in A is the second in the node-pair a. In this case, we assign the appropriate pointers from the current center to a, and we also assign the appropriate pointer (either $\delta(2,1)$ or $\delta(2,2)$) to the current center from a.

In the case where at least one of $\delta(2,1)$ or $\delta(2,2)$ does not point to G, we must create a new node-pair in B. Then we make the appropriate pointer assignments from this new node-pair to the current center and to G.

With the addition of a few extra pointers, we can eliminate H_1, H_2, and G. We simply encode the information these nodes provide with extra pointers

from the origin. For example, we could substitute pointers ϕ_1 and ϕ_2 for ϕ. One of these two would point to the current center node-pair from the origin to indicate whether the center is node 1 or node 2.

If A creates $S(n)$ nodes, then B creates $\lceil S(n)/2 \rceil$ nodes (if we eliminate H_1, H_2, and G). We can then generalize the procedure (or continue to apply it repetitively) to achieve space complexity $cS_m(n)$ for any $c < 1$. □

Note that this simulation does not establish space compression for capacity complexity: if the pointer alphabet size of the original machine is d, then the alphabet size of the simulator is $d(1/c)^2$.

Although space compression is possible using mass as the space measure, it is unclear whether PMs also enjoy the linear speedup property of TMs.

4 Time and Space Hierarchies

We obtain PM time and space hierarchies that parallel hierarchies for RAMs and TMs. We begin with the following useful result:

Lemma 4.1 *For every pointer machine M there is a pointer machine M' with $|\Delta'| = 2$ and $|\Sigma'| = 2$ that simulates M in real time and constant factor overhead in space.*

Proof. Let Δ be the pointer alphabet of M, with $d = |\Delta|$, and let $\Delta' = \{\alpha, \beta\}$ in M'. For each node x in M and each δ in Δ, let $y_\delta = p_\delta(x)$. Corresponding to each node x in M there is a cycle of $d + 1$ nodes in M', connected by α pointers, with one distinguished node x'.

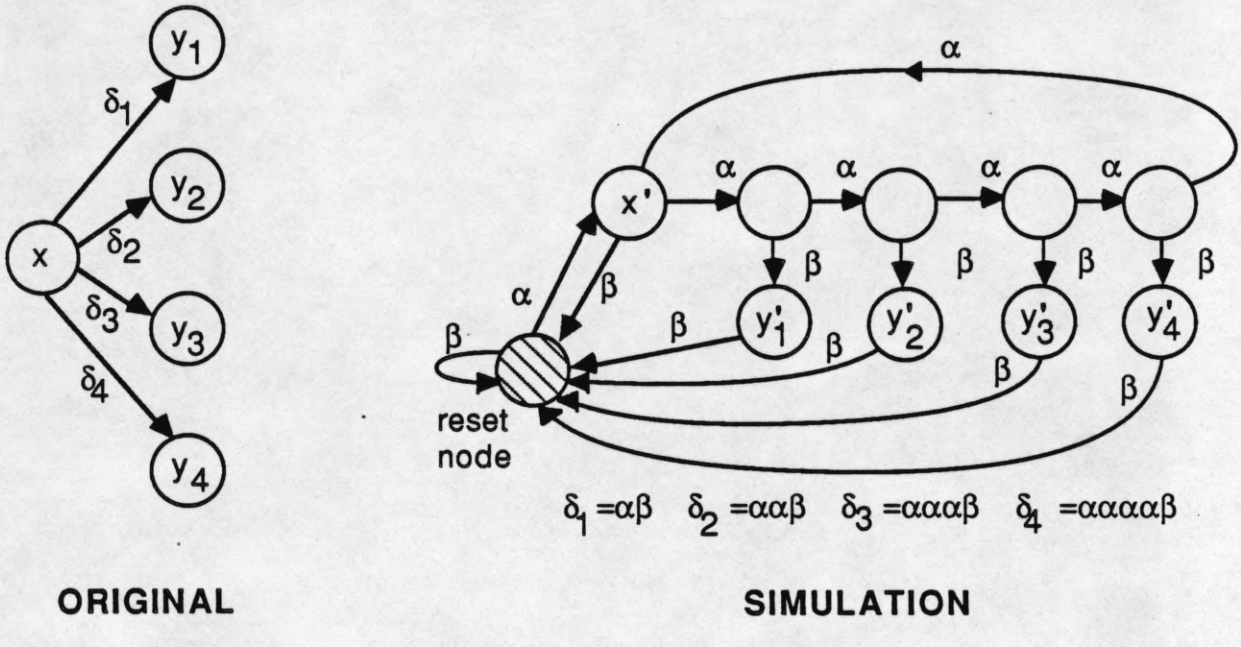


Figure 2: Simulating a PM with $|\Delta'| = 2$.

Consider one such cycle in M'. From each node in the cycle, except x', use the β pointer to point to the appropriate y'_δ corresponding to y_δ. The β pointer from x' points to a special *reset node*. The reset node's α pointer always points to the distinguished node representing the current center. The reset node and the cycle of $d + 1$ nodes are necessary in order to return to the center after simulation of each instruction (see Figure 2).

Simulating the instructions of M with this structure can be somewhat tricky. We give an example in the Appendix to show how M' would simulate one instruction of M.

To show that two symbols are sufficient for Σ', we note that we can use a binary encoding of each symbol in Σ (Hopcroft and Ullman, 1979). □

Theorem 4.2 *If $T_2(n)$ is time-constructible by a pointer machine, then there is a language $A \subseteq \{1,2\}^*$ such that some pointer machine recognizes A within time $O(T_2(n))$, but for any function $T_1(n) = o(T_2(n))$, no pointer machine recognizes A within time $O(T_1(n))$.*

Proof. We use a technique similar to that of Cook and Reckhow (1973). To demonstrate the time hierarchy, we must be able to encode any PM program with {1,2}, so that we can construct a "universal PM." Assume that the pointer alphabet is $\{a_1, a_2, \ldots\}$ and μ is an instruction label. Here is a possible encoding:

accept	122
reject	1^222
create a_i	1^321^i22
center a_i	1^421^i22
assign $a_i := a_ja_k$	$1^521^i21^j21^k22$
if $a_i = a_j$ **go to** μ	$1^621^i21^j21^\mu22$
if *input* $= \sigma_i$ **go to** μ	$1^721^i21^\mu22$
move ρ_i (where $\rho_i \in$ {**left**, **right**})	1^821^i22

By Lemma 4.1 we may assume without loss of generality that for the machine to be simulated, $|\Delta| = 2$ and $|\Sigma| = 2$. Therefore, $\max(i,j,k) = 2$.

As with RAMs (Cook and Reckhow, 1973), a PM program can be encoded by concatenating the encodings of its instructions. Let M_w be the PM whose encoding is w. For every language Z accepted by a PM, and for every integer l, there is a ζ such that $|\zeta| > l$ and M_ζ accepts exactly Z; i.e., $\mathcal{L}(M_\zeta) = Z$:

we simply append a sufficient number of 2's to some shorter word ζ', where $\mathcal{L}(M_{\zeta'}) = Z$.

Define A as follows: if M_w with input w halts in time $T_2(|w|)$, then $w \in A$ if and only if M_w does not accept w. If M_w does not halt in time $T_2(|w|)$, then we do not care whether M_w accepts w.

Now suppose M' recognizes A in time $cT_1(n)$, where c is a constant depending on A. Then there exists a long string w' such that $\mathcal{L}(M_{w'}) = \mathcal{L}(M')$, and, since $T_1 = o(T_2)$, $cT_1(|w'|) < T_2(|w'|)$. By our definition of A, $w' \in A$ (i.e., w' is accepted by M') if and only if w' is not accepted by $M_{w'} = M'$; so we have a contradiction. Therefore no such M' exists.

Now we construct a PM M that recognizes A within time $O(T_2(n))$. On input w, M will decode w, creating a node for each instruction. It will then "walk through" the instructions, simulating the machine M_w.

M goes through three stages: initialization, preprocessing, and simulation.

1. **Initialization:**

M first creates eight "instruction type" nodes (*it-nodes*) with an appropriate *it-pointer* corresponding to each. The it-nodes correspond to the eight PM instruction types mentioned above.

M then creates three "argument" nodes (*a-nodes*) with an appropriate *a-pointer* corresponding to each. M uses the a-nodes to designate appropriate arguments for the PM instructions.

Initialization takes a constant amount of time.

2. **Preprocessing:**

M reads w and decodes it, creating a node for each instruction. Each instruction node has the following additional pointers:

a. a *successor* pointer to the next instruction node.

b. an *instruction type* pointer to the appropriate it-node.

c. at most three *argument* pointers to the appropriate a-nodes.

d. a *jump* pointer, for an **if** instruction, to point to the instruction node to which control could be passed.

e. a *beginner* pointer to the origin (used to set the *jump* pointer appropriately).

Instruction nodes use the it-pointers to make a comparison from any node; i.e., we can use the it-pointer to determine what instruction type this instruction node has. We handle a-pointers in a similar manner.

To set the *jump* pointers, M makes a second pass through the instruction nodes and the input word w. For each node representing an **if** instruction, M returns to the origin with the *beginner* pointer (remembering the current instruction node). Using the unary encoding of M in w, M moves the *jump* pointer to the m^{th} instruction node, where the **if** instruction specifies a conditional jump to instruction m. In this way, setting the *jump* pointers takes time $O(T_2(|w|))$.

During preprocessing, M also creates a linked list of $T_2(|w|)$ nodes to be used as a counter for the simulation. This is possible since T_2 is time-constructible. There is one special pointer to the last node in this list.

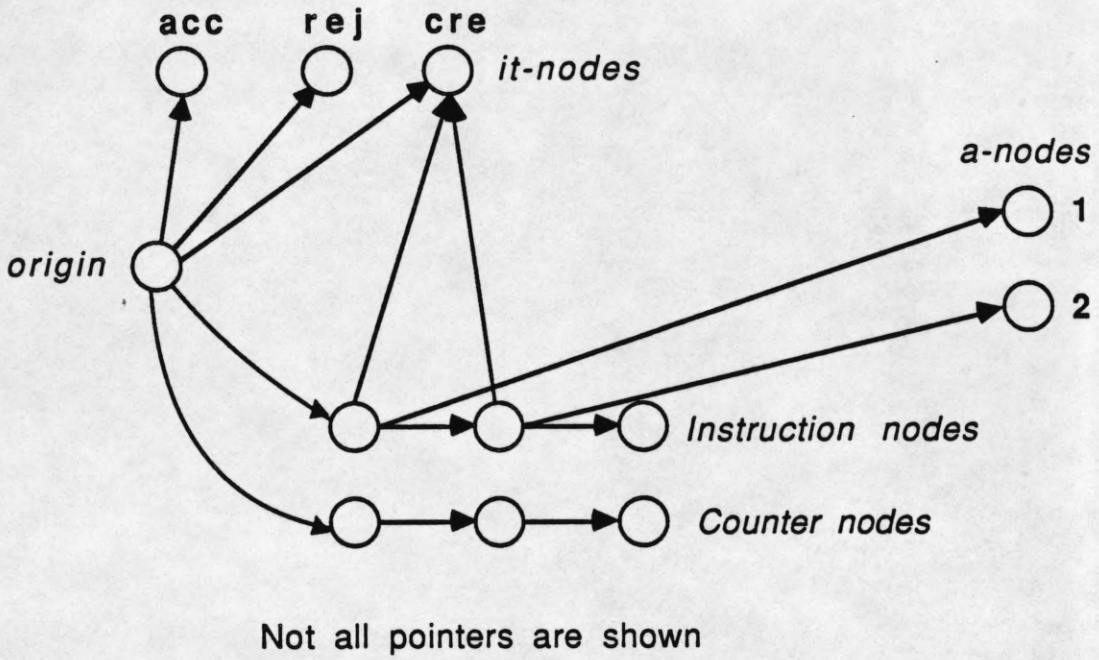


Figure 3: Preparation for Simulating M_w

Two additional pointers keep track of the simulation. The *execute* pointer points from the origin to the node representing the instruction being simulated. The *counter* pointer points to a counter node to indicate how much time has elapsed.

Preprocessing takes $O(T_2(|w|)) + O(|w|) = O(T_2(|w|))$ time (Figure 3 shows the result of initialization and preprocessing).

3. **Simulation:**

The origin serves as the point of reference for the simulation. It is the center node at the beginning of each simulated instruction.

To begin the simulation, M resets the execution and counter pointers to their respective initial nodes. M also resets the input tape head to the leftmost nonblank tape cell. Using the instruction pointer, M performs an instruction by instruction simulation of M_w. After each simulated instruction, M moves to the next instruction node by following the successor or jump pointer, as appropriate.

M uses the counter nodes to keep track of how many instructions have been simulated. After each simulated instruction, M sets the counter pointer to point from the origin to the next counter node in the chain.

Since M has a special pointer to the last counter node, M can compare the special pointer with the counter pointer to determine whether $T_2(|w|)$ steps have elapsed. If so, then M rejects w. Otherwise, M accepts w if and only if M_w rejects w. Therefore, M accepts A in time $O(T_2(|w|))$. □

We now establish a space hierarchy result.

Theorem 4.3 *If $S_2(n)$ is mass-constructible, then there is a language $A \subseteq \{1,2\}^*$ such that some pointer machine recognizes A within mass $O(S_2(n))$, but for any function $S_1(n) = o(S_2(n))$, no pointer machine recognizes A within mass $O(S_1(n))$.*

Proof. As in Theorem 4.2, define A as follows: if M_w with input w halts with mass $S_2(|w|)$, then $w \in A$ if and only if M_w does not accept w. If M_w uses more than $S_2(|w|)$ nodes, then we do not care whether M_w accepts w.

M proceeds as follows: M encodes M_w as in the time hierarchy proof, except it creates $S_2(|w|)$ "counter nodes." Also, M creates only $\min(S_2(|w|), |w|)$ instruction nodes (in case $S_2(n) < n$).

M then runs as before, except it increments the "counter" only when it encounters a **create** instruction. If M runs out of space, then it rejects w. Otherwise, M accepts w if and only if M_w rejects w. Therefore M accepts A in space $O(S_2(n))$.

But what if M_w rejects by not halting? We apply the "backward simulation" technique of Sipser (1980). By applying Sipser's technique, we see that, as with Turing machines, for PM M there is a PM N accepting the same language as M using the same number of nodes, but N does not loop on a finite number of nodes. To prove this, we would simply replace all occurrences of tapes in Sipser's proof with sets of nodes. □

Note that the space hierarchy also applies when we use capacity as the space measure.

Corollary 4.4 *If $S_2(n)$ is capacity-constructible, then there is a language $A \subseteq \{1,2\}^*$ such that some pointer machine recognizes A within capacity $O(S_2(n))$, but for any function $S_1(n) = o(S_2(n))$, no pointer machine recognizes A within capacity $O(S_1(n))$.*

Corollary 4.5 *$TIME_{PM}(T)$ is strictly included in $MASS(T)$.*

Proof.

Halpern *et al.* (1986) showed that $\text{TIME}_{PM}(T) \subseteq \text{MASS}(O(T/\log T))$.
By Theorem 3.1, $\text{MASS}(O(T/\log T)) \subseteq \text{MASS}(T/\log T)$.
By Theorem 4.3, $\text{MASS}(T/\log T) \subset \text{MASS}(T)$. □

5 Space Requirements and the Invariance Thesis

Slot and van Emde Boas (1988) proposed the following *Invariance Thesis*:

> "There exists a standard class of machine models, which includes among others all variants of Turing machines, all variants of RAMs and RASPs with logarithmic time and space measures, and also the RAMs and RASPs in the uniform time and logarithmic space measure, provided only standard arithmetical instructions of additive type are used. Machine models in this class simulate each other with polynomially bounded overhead in time and constant factor overhead in space."

An obvious question is, does this thesis apply as well to PMs? Schönhage (1980) presented a real-time equivalence between "successor RAMs," which meet the qualifications of the thesis with respect to time, and PMs. So the thesis holds for PMs with respect to time complexity. For space complexity, however, the equivalence depends on the space measure used. In the following discussion, $\text{space}_{TM}(S)$ will denote Turing machine space and $\text{space}_{RAM}(S)$ will denote log cost RAM space. Log cost RAM space is defined as the sum

over all nonempty registers of the logarithm of the largest integer stored in each register (Slot and van Emde Boas, 1988).

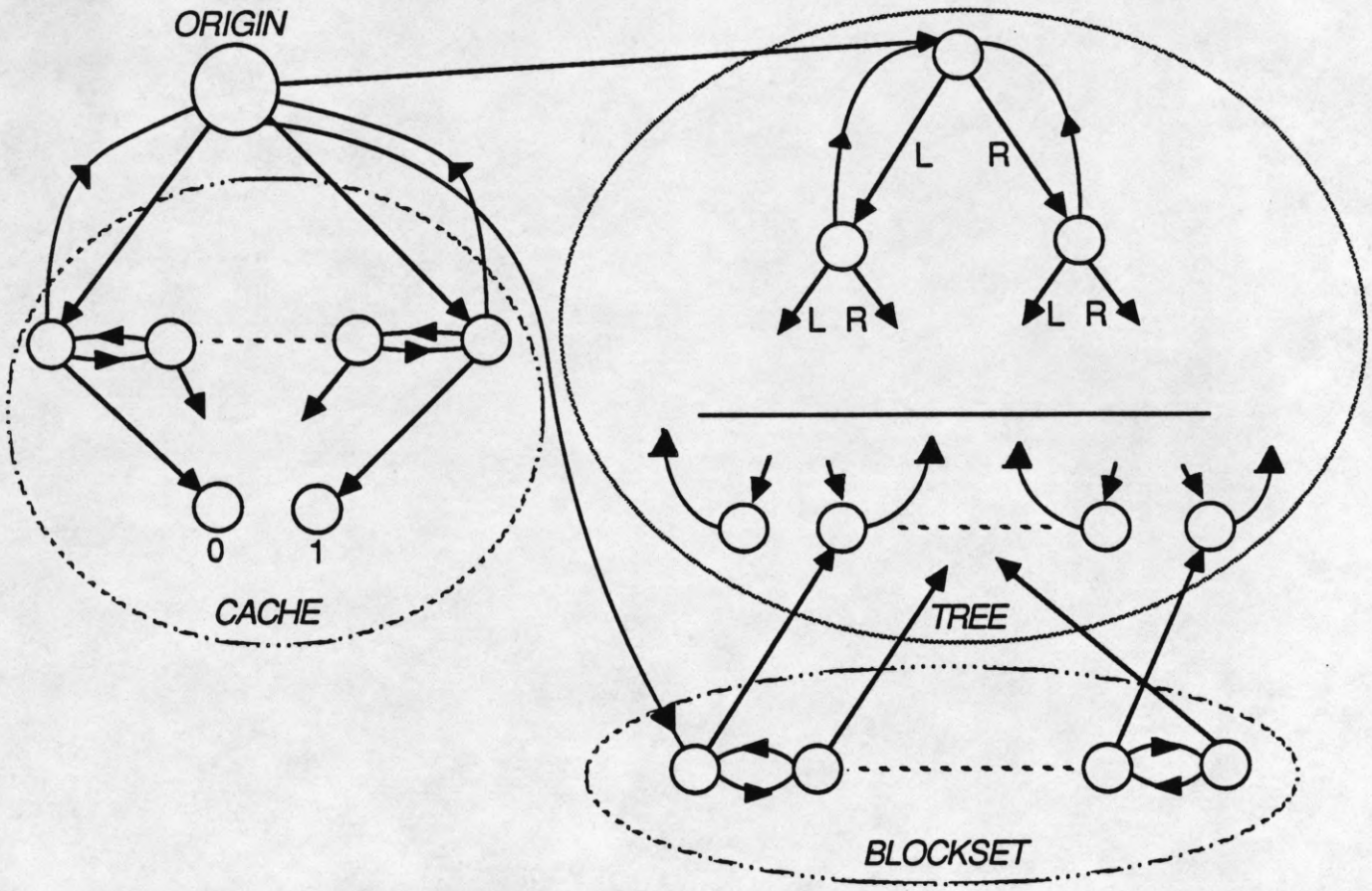
Theorem 5.1 *A multitape Turing machine using space $S(n)$ and time $T(n)$ can be simulated by a pointer machine using $O(S(n)/\log S(n))$ nodes and time $O(T(n))$. Hence, a pointer machine uses less space than a Turing machine accepting the same language, if we consider mass as the space measure for pointer machines.*

Proof. We may assume that S is mass-constructible. If it is not, then we can try $S = 1, 2, 4 \ldots$ until the simulation succeeds (van Emde Boas, 1987).

The following simulation holds for TMs with multiple worktapes; however, for simplicity, we will explain how to simulate a TM with a single one-way infinite worktape and a read-only input tape. The worktape alphabet of the TM is $\{0, 1\}$.

We design a one-to-one correspondence between the storage configurations of a TM M using space S and the configurations of a PM M' with $O(S/\log S)$ nodes: we partition the worktape of M into blocks of size $b = \lg(S/\lg S)$. With this partitioning, there are $S/b = O(S/\log S)$ blocks. M' represents the tape contents with three node structures: the *tree*, the *blockset*, and the *cache* (see Figure 4).

The *tree* is a complete binary tree of height b. The *blockset* consists of $O(S/\log S)$ nodes, $\beta_0, \beta_1, \ldots$, each node representing one block. The contents of a particular block are represented by a pointer to a leaf of the tree. Since the tree has $2^b = S/\lg S$ leaves, there is a one-to-one correspondence between the leaves of the tree and the contents of a block.



Additional pointers include a pointer from every node to the origin

Figure 4: Representing a Turing machine with pointer machine nodes

Now consider two adjacent blocks B_i and B_{i+1} of M, such that the worktape head is currently in either B_i or B_{i+1}. B_i and B_{i+1} are represented in the blockset by β_i and β_{i+1}, respectively. During the simulation, M' keeps the contents of β_i and β_{i+1} in the cache.

The *cache* consists of a chain of $2\lg(S/\lg S)$ nodes and two additional nodes, "0" and "1." Each node in the chain has a pointer to either the "0" or "1" node, so that the entire chain is a direct representation of two blocks of M's worktape.

M' decodes a node β_i into the cache as follows: M' finds the tree leaf pointed to by β_i. M' then traces the path of the tree to the root, noting at each tree node whether it was a right or left child. For each tree node in the path, M' sets a pointer of a node in the cache to the "0" or "1" node, depending on whether the tree node was a right or left child.

M' encodes the contents of half the cache back to the blockset by following the above steps in reverse.

The simulation proceeds as follows: M' initially builds the blockset, tree, and cache. We assume blocks are numbered from left to right, and that M starts with its worktape head on the leftmost tape cell. So M' initially decodes β_0 and β_1 into the cache. M' then begins the actual simulation of M.

Assume β_i and β_{i+1} are decoded in the cache. As long as the tape head remains in B_i and B_{i+1}, M' performs a straightforward simulation of M, using the cache. Finite control and input processing of M are simulated in a straightforward manner using finite control and input processing of M'. When the tape head moves to the right of B_{i+1}, M' encodes B_i back to β_i

and decodes β_{i+2} into the cache, shifting to the left the "contents" of B_{i+1} currently in the cache. If the tape head moves to the left of B_i, then a similar operation occurs. At this time, since the worktape head of M is in the middle of two blocks, M' can simulate at least b steps of M before performing the encoding and decoding operations again.

During the simulation, M' creates $O(S/\log S)$ nodes: $O(S/\log S)$ nodes for the tree and blockset and $O(\log(S/\log S))$ nodes for the cache. Therefore M' simulates M with mass $O(S/\log S)$.

The straightforward simulation in the cache requires a total of $O(T)$ time. The only other time requirement is for encoding and decoding the blocks. Encoding and decoding can be done in $O(b)$ time, since that is the height of the tree. Because M' maintains two decoded blocks, it performs an encoding and decoding only every $O(b)$ steps, so the total time required for the simulation is $O(T)$.

The entire simulation requires $O(S/\log S)$ nodes. Therefore, we have $\text{SPACE}_{TM}(S) \subseteq \text{MASS}(O(S/\log S)) \subset \text{MASS}(S)$, by Theorem 4.3, so less space is required for a PM than for a TM. □

Theorem 5.1 improves the result of van Emde Boas (1987), whose PM simulator used space $O(S/\log\ S)$, but time $O(T^2)$.

But is the simulation of Theorem 5.1 optimal? More specifically, can we simulate a TM using space S with $o(S/\log S)$ nodes? Before we answer that, let us consider the validity of the Invariance Thesis if we use the the capacity space measure for PMs. This will help us answer our question about optimality of the simulation.

Theorem 5.2 *If we use the capacity space measure for pointer machines, then a RAM can simulate a pointer machine with constant factor overhead in space, and vice-versa.*

Proof. For classes of machines X and Y, write $X \stackrel{c}{\Longrightarrow} Y$ if for every machine M_Y in Y there exists a machine M_X in X such that M_X simulates M_Y with constant factor overhead in space.

To simulate a PM using capacity S with a RAM using space $O(S)$, we simply use Schönhage's (1980) technique of simulating a PM by a successor RAM . Register 1 of the RAM contains the address of the center, Register 2 contains the address of the first available free register for creating new nodes, and the next few registers contain some work space. Beyond these registers, each node of the PM is represented by d consecutive registers of the RAM, where d is the size of the pointer alphabet. Each register represents a pointer δ and contains an address of a register corresponding to the node pointed to by δ. So a PM with S' nodes ($O(S' \log S')$ capacity) can be simulated by a RAM with $O(S')$ registers, i.e., in $\text{space}_{RAM} O(S' \log S')$. Therefore, $RAM \stackrel{c}{\Longrightarrow} PM$.

By Theorem 5.1, every TM using space S can be simulated by a PM with $O(S/\log S)$ nodes, which uses capacity $O((S/\log S)\log(S/\log S)) = O(S)$. Thus $PM \stackrel{c}{\Longrightarrow} TM$. Slot and van Emde Boas (1988) showed $TM \stackrel{c}{\Longrightarrow} RAM$, so by transitivity of $\stackrel{c}{\Longrightarrow}$, $PM \stackrel{c}{\Longrightarrow} RAM$. □

Corollary 5.3 *A Turing machine can simulate a pointer machine with constant factor overhead in space, and vice-versa.*

We now have space equivalence for TMs, RAMs, and PMs. We show that this equivalence implies that a PM must use $\Omega(S/\log S)$ nodes to simulate a Turing machine of space $O(S)$. Assume, to the contrary, that we can simulate a TM of space $O(S)$ with $o(S/\log S)$ nodes. This implies we can simulate the TM by a PM in $o(S)$ capacity. By Corollaries 4.4 and 5.3, $\text{SPACE}_{TM}(S) \subseteq \text{CAPACITY}(o(S)) \subset \text{CAPACITY}(S) \subseteq \text{SPACE}_{TM}(O(S))$, so $\text{SPACE}_{TM}(S)$ is strictly included in $\text{SPACE}_{TM}(S)$, a contradiction.

If we consider capacity as the true measure of space in PMs, then we must reevaluate the result of Halpern *et al.* (1986): that every PM of time complexity T can be simulated by a PM of space complexity $O(T/\log T)$. The authors considered mass as the space measure. A different approach is necessary to achieve the same result for capacity, if indeed it is even possible.

6 Conclusions

Pointer machines are definitely a departure from the more classical models of computation. In particular, PMs are able to alter their storage structures, unlike TMs and RAMs. There is also evidence that PMs are faster than TMs or RAMs (Schönhage, 1980). In many respects, however, the pointer machine behaves like the more classical models of computation. We have shown a space compression result, and we have seen that the deterministic time and space hierarchies remain valid for PMs, although the PM time hierarchy (Theorem 4.2) is sharper than the TM time hierarchy (Hopcroft and Ullman, 1979).

"Equivalence" of PMs with the other models does not hold if we use the "obvious" space measure (mass) (Halpern *et al.*, 1986; van Emde Boas, 1987). In this case, we can do more on a PM with the same amount of space, thus violating the Invariance Thesis.

Slot and van Emde Boas (1988) address the question of what to do when the Invariance Thesis is contradicted. Their answer is to adjust the definitions: "The thesis becomes a guiding rule for specifying the right class of models rather than an absolute truth, ..." We have shown that if capacity is adopted as the space measure for PMs, then this machine model fits into the class of models mentioned in the Invariance Thesis.

There are many areas of further research related to this machine model; for instance:

(1) Is linear speedup in time possible on PMs?

(2) What can be said about nondeterministic and parallel PMs? (Gurevich (1988) makes an interesting statement about a model similar to the PM, the KU machine (Kolmogorov and Uspenskii, 1958): "Doesn't the human brain resemble somewhat a parallel KU machine?" What are the implications of such a statement?)

Appendix

The simulation of M by M' in Lemma 4.1

As an example of how M' simulates M, we give the instructions in M' that simulate an **if** instruction. Assume M has $\Delta = \{\delta_1, \delta_2, \delta_3, \delta_4\}$ (refer to Figure 2) and x' is the current center. Call the nodes in the cycle (except for x') auxiliary nodes.

We describe the simulation for the following piece of M's code:

ϕ_1: **if** $\delta_1 = \delta_3$ **go to** μ
ϕ_2:
...
μ:

We describe the simulation at a "macro" level:

ϕ_1: **center** β	(* reset node is now center *)
assign β to point to δ_2 auxiliary node	(* to keep cycle intact *)
make δ_1 auxiliary node the center	
assign α to point to y_3'	
if $\alpha = \beta$ **go to** ψ	(* this means $\delta_1 = \delta_3$ *)
(* otherwise, "clean up" and continue *)	
assign α to point to δ_2 auxiliary node	
make reset node the center	

assign β to point to reset node

center α (* make x' the center *)

if $\alpha = \alpha$ **go to** ϕ_2 (* unconditional jump *)

(* same "clean-up" as above *)

ψ: assign α to point to δ_2 auxiliary node

make reset node the center

assign β to point to reset node

center α (* make x' the center *)

if $\alpha = \alpha$ **go to** μ (* unconditional jump *)

ϕ_2:

...

μ:

References

BORODIN, A., FISCHER, M. J., KIRKPATRICK, D. G., LYNCH, N. A., AND TOMPA, M. (1981), A time-space tradeoff for sorting on non-oblivious machines, *J. Comput. System Sci.* **22**, 351–364.

COOK, S. A., AND RECKHOW, R. A. (1973), Time bounded random access machines, *J. Comput. System Sci.* **7**, 354–375.

GUREVICH, Y. (1988), On Kolmogorov machines and related issues, *Bull. of EATCS.*

HALPERN, J. Y., LOUI, M. C., MEYER, A. R., AND WEISE, D. (1986), On time versus space III, *Math. Systems Theory* **19**, 13–28.

HOPCROFT, J. E., AND ULLMAN, J. D. (1979), "Introduction to Automata Theory, Languages, and Computation", Addison-Wesley, Reading, MA.

KOLMOGOROV, A., AND USPENSKII, V., 1958, On the definition of an algorithm, *Uspekhi Mat. Nauk* **13**, 3–28. English translation in *Amer. Math. Soc. Translations*, ser. 2, vol. 29 (1963), 217-245.

SCHÖNHAGE (1980), Storage modification machines, *SIAM J. Comput.* **9**, 490–508.

SIPSER, M. (1980), Halting space-bounded computations, *Theoret. Comput. Sci.* **10**, 335–338.

SLOT, C., AND VAN EMDE BOAS, P. (1988), The problem of space invariance for sequential machines, *Inform. and Comput.* **77**, 93–122.

TARJAN, R. E. (1979), A class of algorithms which require nonlinear time to maintain disjoint sets, *J. Comput. System Sci.* **18**, 110–127.

VAN EMDE BOAS (1987), "Space Measures for Storage Modification Machines", University of Amsterdam, Tech. Rep. FVI-UvA-87-16. To appear in *Informat. Process. Lett.*.